Data Structures and Algorithms Formula Sheet

Fady Morris Ebeid

(2024)

Chapter 1
Intro

1 Series

Geometric series:

$$\sum_{i=0}^{n-1} ar^i = a + ar + ar^2 + \ldots + ar^{n-1}$$
$$= a\frac{1-r^n}{1-r}$$

Order of growth of a geometric series:

$$\sum_{i=0}^{n} r^i = \begin{cases} \Theta(r^n) & \text{if } r > 1, \\ \Theta(n) & \text{if } r = 1, \\ \Theta(1) & \text{if } r < 1 \end{cases}$$

Arithmetic series:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$
$$= \frac{n(n+1)}{2}$$

Harmonic Series:

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n}$$
$$= \ln n + \gamma$$

Where $\gamma \approx 0.577$ is the Euler-Mascheroni constant

2 Complexity Comparison

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec n^3 \prec 2^n$$

3 Divide and Conquer

3.1 Master Theorem

Theorem 3.1 (Master Theorem)**.** If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O\left(n^d\right)$ (for consants $a > 0, b > 1, d \geq 0$), then:

$$T(n) = \begin{cases} O\left(n^d\right) & \text{if } d > \log_b a \\ O\left(n^d \log n\right) & \text{if } d = \log_b a \\ O\left(n^{\log_b a}\right) & \text{if } d < \log_b a \end{cases}$$

3.2 Binary Search

Complexity

To search for a key, the algorithm makes a single recursive call for a problem of size $n/2$. Outside this call it spends time $O(1)$. Therefore $a = 1, b = 2, d = 0$.

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Unwinding the recurrence relations:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + 2c \\ &= T(n/8) + 3c \\ &\vdots \\ &= T(n/2^k) + kc \\ &\vdots \\ &= T(1) + \log_2 n \cdot c \\ &= O(\log n) \end{aligned}$$

Recursive Version

Algorithm 1: BinarySearch(A[1...n], low, high, key)

```
Data: Sorted Sequence A[1...n], low, high, key
Result: Index of key, -1 if not found
if high < low:
    return low -1
mid ← ⌊low + (high - low)/2⌋
if key = A[mid]:
    return mid
elif key < A[mid]:
    return BinarySearch(A, low, mid - 1, key)
else:
    return BinarySearch(A, mid + 1, high, key)
```

Iterative Version

Algorithm 2: BinarySearchIt(A[1...n], key)

```
Data: Sorted Sequence A[1...n], key
Result: Index of key, -1 if not found
low ← 1
high ← n
while low ≤ high:
    mid ← ⌊low + (high - low)/2⌋
    if key = A[mid]:
        return mid
    elif key < A[mid]:
        high = mid - 1
    else:
        low = mid + 1
return low - 1
```

Binary Search with Duplicates

Algorithm 3: BinarySearch(A[1...n], key)

```
Data: Sorted Sequence A[1...n], key
Result: Index of key, -1 if not found
low ← 1
high ← n
while low < high:
    mid ← ⌊(low + high)/2⌋
    if key <= A[mid]:
        high = mid
    else:
        low = mid + 1
if A[left] == key:
    return low
else:
    return -1
```

3.3 Sorting

Merge Sort

Complexity

The algorithm breaks an array of length n into two subarrays of size $n/2$ and sorts them recursively, then merges the output. Time spent before and after the recursive calls is $O(n)$. Therefore $a = 2, b = 2, d = 1$.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Unwinding:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= 2\left(2T\left(\frac{n}{4}\right) + c\cdot\frac{n}{2}\right) + cn \;= 4T\left(\frac{n}{4}\right) + 2cn \\
&= 4\left(2T\left(\frac{n}{8}\right) + c\cdot\frac{n}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&\vdots \\
&= nT(1) + \log_2 n \cdot cn \\
&= O(n \log n)
\end{aligned}
$$

Algorithm

Algorithm 4: Merge Sort(A[1 … n])

Data: Sequence $A[1\dots n]$
Result: Permutation $A'[1\dots n]$ of A in non-decreasing order

if $n = 1$**:**
  return A
$m \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
$B \leftarrow$ `MergeSort`$(A[1\dots m])$
$C \leftarrow$ `MergeSort`$(A[m+1\dots n])$
$A' \leftarrow$ `Merge`(B, C)
return A'

Algorithm 5: Merge(B[1…p], C[1…q])

Data: Sequences $B[1\dots p], C[1\dots q]$
`/* B and C re sorted */`
$D \leftarrow$ empty array of size $p + q$
while *B and C are both non-empty***:**
  $b \leftarrow$ the first element of B
  $c \leftarrow$ the first element of C
  if $b \leq c$**:**
    move b from B to the end of D
  else:
    move c from C to the end of D
move the rest of B and C to the end of D
return D

Count Sort

Is a non-comparison based sorting algorithm.
Useful if the array contents are small integers that have large frequencies.
The complexity is $O(n + M)$.

Algorithm 6: CountSort(A[1…n])

Data: A[1 … n] with elements that are all integers from 1 to M
Result: A'[1 … n]

Count[1 … M] ← [0, … , 0]
for *i from 1 to n***:**
  Count[A[i]] ← Count[A[i]] + 1
`/* k appears Count[k] times in A */`
Pos[1 … M] ← [0, … , 0]
Pos[1] ← 1
for *j from 2 to M***:**
  Pos[j] ← Pos[j - 1] + Count[j - 1]
`/* k will occupy range [Pos[k]...Pos[k + 1] - 1] */`
for *i from 1 to n***:**
  A'[Pos[A[i]]] ← A[i]
  Pos[A[i]] ← Pos[A[i]] + 1
return *A'*

Quick Sort

Average case: $O(n \log n)$

Worst case: $O(n^2)$

Algorithm 7: QuickSort(A[1…n], ℓ, r)

Data: $A[1\dots n], \ell, r$
if $\ell \geq r$**:**
  return
$m \leftarrow$ `Partition`(A, ℓ, r)
`/* A[m] is in the final position */`
`QuickSort`$(A, \ell, m-1)$
`QuickSort`$(A, m+1, r)$

Algorithm 8: Partition2(A[1…n], ℓ, r)

Data: $A[1\dots n], \ell, r$
Result: j
$x \leftarrow A[\ell]$ `/* pivot */`
$j \leftarrow \ell$
for i *from* $\ell + 1$ *to* r**:**
  if $A[i] \leq x$**:**
    $j \leftarrow j + 1$
    `swap` $A[j]$ and $A[i]$
  `/*` $A[\ell+1\dots j] \leq x, A[j+1\dots r] > x$ `*/`
`swap` $A[\ell]$ and $A[j]$
return j

Randomized QuickSort

Algorithm 9: RandomizedQuickSort(A, ℓ, r)

Data: $A[1\dots n], \ell, r$
if $\ell \geq r$**:**
  return
$k \leftarrow$ random number between ℓ and r
`swap` $A[\ell]$ and $A[k]$
$m \leftarrow$ `Partition`(A, ℓ, r)
`/*` $A[m]$ `is in the final position */`
`RandomizedQuickSort`$(A, \ell, m-1)$
`RandomizedQuickSort`$(A, m+1, r)$

QuickSort with Tail Recursion Elimination:

Algorithm 10: QuickSort(A[1…n], ℓ, r)

Data: $A[1\dots n], \ell, r$
while $\ell < r$**:**
  $m \leftarrow$ `Partition`(A, ℓ, r)
  if $(m - \ell) < (r - m)$**:**
    `QuickSort`$(A, \ell, m-1)$
    $\ell \leftarrow m + 1$
  else:
    `QuickSort`$(A, m+1, r)$
    $r \leftarrow m - 1$

Worst-case space requirements: $O(\log n)$.

Randomized QuickSort with Equal Elements

Sort a given sequence of numbers that may contain duplicates.

Algorithm 11: RandomizedQuickSort(A, ℓ, r)

Data: $A[1\dots n], \ell, r$
if $\ell \geq r$**:**
  return
$k \leftarrow$ random number between ℓ and r
`swap` $A[\ell]$ and $A[k]$
$(m_1, m_2) \leftarrow$ `Partition3`(A, ℓ, r)
`/*` $A[m]$ `is in the final position */`
`RandomizedQuickSort`$(A, \ell, m_1 - 1)$
`RandomizedQuickSort`$(A, m_2 + 1, r)$

Algorithm 12: Partition3(A[1...n], ℓ, r)

```
Data: A[1...n], ℓ, r
Result: m2
x ← A[ℓ] /* pivot */
m1 ← ℓ
m2 ← ℓ
for i from ℓ+1 to r:
    if A[i] ≤ x:
        m2 ← m2 + 1
        swap A[i] and A[m2]
    if A[m2] < A[m1]:
        swap A[m1] and A[m2]
        m1 ← m1 + 1
    /* A[ℓ+1...m1] ≤ x, A[m2+1...r] > x */
return (m1, m2)
```

4 Dynamic Programming

4.1 Edit Distance

[Eri19, p. 129] [KP18, p. 195]

The *Edit* function satisfies the following recurrence:

$$\text{Edit}(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \left\{ \begin{array}{l} \text{Edit}(i, j-1) + 1 \\ \text{Edit}(i-1, j) + 1 \\ \text{Edit}(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

Complexity: $O(mn)$ space and time.

Algorithm 13: EditDistance(A[1..m], B[1..n])

```
for j ← 0 to n:
    Edit[0, j] ← j
for i ← 1 to m:
    Edit[i, 0] ← i
    for j ← 1 to n:
        ins ← Edit[i, j-1] + 1
        del ← Edit[i-1, j] + 1
        rep ← Edit[i-1, j-1]
        if A[i] ≠ B[j]:
            rep ← rep + 1
        Edit[i, j] ← min{ins, del, rep}
return Edit[m, , n]
```

4.2 Longest Common Subsequence

[KP18, p. 205]

$$\text{LCS}(j,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{LCS}(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{otherwise} \end{cases}$$

Complexity: $O(mn)$ space and time.

Algorithm 14: GetLCS2(A[1..m], B[1..n])

```
for j ← 0 to n:
    LCS[0, j] ← 0
for i ← 1 to m:
    LCS[i, 0] ← 0
    for j ← 1 to n:
        if A[i] = B[j]:
            LCS[i, j] ← LCS[i-1, j-1] + 1
        else:
            LCS[i, j] ← max{LCS[i, j-1], LCS[i-1, j]}
return LCS[m, n]
```

4.3 Knapsack

Knapsack with Repititions

Algorithm 15: Knapsack(W, weights[$w_1 \dots w_n$], vals[$v_1 \dots v_n$])

```
Data: Weights w1, ..., wn, values v1, ..., vn, and total
      weight W
Result: The maximum value of items whose weight
        doesn't exceed W
value[0] ← 0
for w from 1 to W:
    value[w] ← 0
    for i from 1 to n:
        if wi ≤ w:
            val ← value[w − wi] + vi
            if val > value[w]:
                value[w] ← val
return value[W]
```

Knapsack without Repititions

Subproblems:

$$\text{value}[w,i] = \max\{\text{value}[w - w_i, i-1] + v_i, \text{value}[w, i-1]\}$$

Running time: $O(nW)$

Algorithm 16: Knapsack(W, weights[$w_1 \dots w_n$], vals[$v_1 \dots v_n$])

```
Data: Weights w1, ..., wn, values v1, ..., vn, and total
      weight W
Result: The maximum value of items whose weight
        doesn't exceed W. Each item can be used at most
        once.
initialize all value[0, j] ← 0
initialize all value[w, 0] ← 0
for i from 1 to n:
    for w from 1 to W:
        value[w, i] ← value[w, i-1]
        if wi ≤ w:
            val ← value[w − wi, i − 1] + vi
            if value[w, i] < val:
                value[w, i] ← val
return value[W, n]
```

4.4 Placing Parentheses

[KP18, p. 226]

Example:

How to place parentheses to maximize the expression $5 - 8 + 7 \times 4 - 8 + 9$.

Solution:

The maximum is 200

Given by $5 - (8 + 7) \times (4 - (8 + 9))$

Subproblems:

Let $E_{i,j}$ be the subexpression

$$d_i \text{ op}_i \dots \text{op}_{j-1}\, d_j$$

$M(i,j) =$ Maximum value of $E_{i,j}$

$m(i,j) =$ Minimum value of $E_{i,j}$

$$M(i,j) = \max_{i \le k \le j-1} \begin{cases} M(i,k) & \text{op}_k & M(k+1,j) \\ M(i,k) & \text{op}_k & m(k+1,j) \\ m(i,k) & \text{op}_k & M(k+1,j) \\ m(i,k) & \text{op}_k & m(k+1,j) \end{cases}$$

$$m(i,j) = \min_{i \le k \le j-1} \begin{cases} M(i,k) & \text{op}_k & M(k+1,j) \\ M(i,k) & \text{op}_k & m(k+1,j) \\ m(i,k) & \text{op}_k & M(k+1,j) \\ m(i,k) & \text{op}_k & m(k+1,j) \end{cases}$$

Running time: $O(n^3)$

Algorithm 17: MinAndMax(i,j)

```
Data: M, m : 2D Matrices holding the maximum and
      minimum values, respectively.
min ← +∞
max ← −∞
for k from i to j − 1:
    a ← M(i,k)  op_k  M(k+1,j)
    b ← M(i,k)  op_k  m(k+1,j)
    c ← m(i,k)  op_k  M(k+1,j)
    d ← m(i,k)  op_k  m(k+1,j)
    min ← min(min, a, b, c, d)
    max ← max(max, a, b, c, d)
return (min, max)
```

Algorithm 18: Parentheses(d_1 op$_1$ $d_2 \ldots$ op$_{n-1}$ d_n)

```
Data: A sequence of digits d_1, ... d_n and a sequence of
      operations op_1, ... op_n ∈ {+, −, ×}
Result: Maximum value that can be obtained by optmial
        parenthesizing of the expression
for i from 1 to n:
    m(i,i) ← d_i, M(i,i) ← d_i
for s from 1 to n − 1:
    for i from 1 to n − s:
        j ← i + s
        m(i,j), M(i,j) ← MinAndMax(i,j)
return M(1,n)
```


References

[Eri19] J. Erickson. *Algorithms*. Jeff Erickson, 2019. ISBN: 9781792644832. URL: https://books.google.com.eg/books?id=K1uIxwEACAAJ.

[KP18] Alexander S Kulikov and Pavel Pevzner. *Learning Algorithms Through Programming and Puzzle Solving*. Active Learning Technologies, 2018. ISBN: 9780985731212. URL: https://cogniterra.org/a/24.